Lecture Notes CS:5350 Design and Analysis of Algorithms
Lecture : November 14, 2019
Scribe: Ingroj Shrestha

1 Revisiting SET COVER

In this section, we will discuss SET COVER problem.

We are given a ground set X containing m elements i.e., $|X| = m$. We are also given a list $S_1, S_2, ..., S_n$ of subsets of X.

DEFINITION: A SET COVER is a collection of sets whose union is equal to all the ground set X.

We will assume that $\bigcup_{i=1}^{n} S_i = X$.

QUESTION: What is the fewest sets that can cover X ?

INPUT: $X, S_1, S_2, S_3, ..., S_n$

OUTPUT:
Let us consider set of indices C such that

$$C \subseteq \{1, 2, 3, ..., n\},$$

We want to find a subset of the index set C such that if we take the union over the subsets indexed by these indices, then the union will cover the entire set X and the size of C is minimum. This can be represented as below:

$$\bigcup_{i \in C} S_i = X$$

and $|C|$ is minimum.

2 Greedy Heuristic for SET COVER

In this section, we formalize a greedy algorithm for the SET COVER problem.

In a greedy approach to the SET COVER problem, in each stage, we select the set that covers the greatest number of uncovered elements. The pseudocode for the algorithm is as below:

$u \leftarrow X$ // u is set of uncovered elements
$C \leftarrow \phi$ // C is set of indices of the cover
while $u \neq \phi$ **do**
  Pick S_i such that $|S_i \bigcap u|$ is maximum. //Greedy Step
  Add i to C
  $u \leftarrow u \backslash S_i$
output C

Algorithm 1: Greedy Algorithm for SET COVER

The set u represents the set of uncovered elements, and the set C represents the set of indices of the cover. At the beginning of the algorithm, none of the elements is covered. So, we set u to X, and C to an empty set. We repeatedly select the set S such that it covers the greatest number of uncovered elements of X, and add indices of the selected subset to C.

EXAMPLE

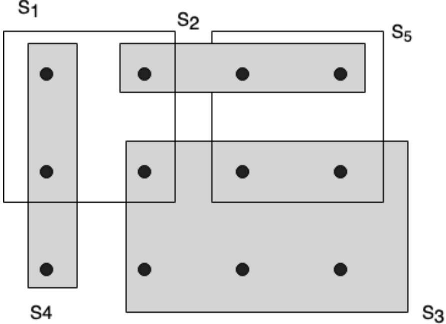

Figure 1: SET COVER

In this Figure1, the greedy algorithm will select set S_3 as it covers the greatest number of uncovered elements, i.e., 6. The algorithm then selects either of sets S_1, S_2, or S_4 randomly as each of these sets contains the same number of uncovered elements. The process is repeated until all of the elements in X are covered. The greedy heuristic will provide S_3, S_4, and S_2 as the fewest sets that cover all of the uncovered elements.

3 Analyzing Greedy Algorithm for SET COVER

In this section, we will discuss the following theorem.

Theorem: The greedy algorithm for SET COVER produces a cover C such that

$$|C| \leq O(\log m).|C^*|$$

i.e., the greedy algorithm is an order $\log m$ approximation.

where, $m = |X|$ is the size of ground set, C is a greedy set cover, $|C|$ is the size of greedy set cover, C^* is an optimal set cover, and $|C^*|$ is the size of the optimal set cover.

PROOF

Let, $g = |C|$ be the size of greedy set cover, and $c^* = |C^*|$ be the size of optimal set cover.

Let us consider the number of elements we cover in each iteration of the algorithm. Let m_i denote the number of elements yet to be covered after iteration i.

$\therefore m_0 = m = |X|$

Also, after g iteration, the the number of elements yet to be covered i.e., $m_g = 0$

Consider the situation after iteration $i-1$. After iteration $i-1$, the number of elements yet to be covered $(u) = m_{i-1}$. We know that the optimal set cover covers all of X, and therefore it covers u at this stage also. Therefore, there is at least one set in the optimal solution that covers at least $\frac{m_{i-1}}{c^*}$ elements.

By pigeonhole principal, in iteration i, the greedy algorithms, therefore, picks a set that covers at least $\frac{m_{i-1}}{c^*}$ elements of u. Since greedy algorithm picks the set with the greatest number of uncovered elements, it must pick a set that covers at least $\frac{m_{i-1}}{c^*}$ elements.

Then, we obtain the number of elements to be covered after iteration i i.e., m_i as:

$m_i \leq m_{i-1} - \frac{m_{i-1}}{c^*}$

$\therefore m_i = \left(1 - \frac{1}{c^*}\right) m_{i-1}$

$\therefore m_1 \leq \left(1 - \frac{1}{c^*}\right) m_0$

$\quad \leq \left(1 - \frac{1}{c^*}\right) m$

$\therefore m_2 \leq \left(1 - \frac{1}{c^*}\right) m_1$

$\quad \leq \left(1 - \frac{1}{c^*}\right)^2 m$

$\therefore m_g \leq \left(1 - \frac{1}{c^*}\right)^g m$

Using the equation $1 + x \leq e^x$, we get
$\quad \leq e^{\frac{-g}{c^*}}.m$

If we pick g large enough to make RHS less than 1, the algorithm terminates as there will not be elements left to be covered.

Now, we solve: $e^{\frac{-g}{c^*}}.m = 1$

$m = e^{\frac{g}{c^*}}$

Taking natural logarithm on both sides we get,

$\ln m = \frac{g}{c^*}$

$\therefore g = c^* \ln m$

When g exceeds $c^* \ln m$, the algorithm terminates.

4 Revisiting Greedy Algorithm Analysis

Recall the theorem from the last lecture,

Theorem: For any $\epsilon > 0$, there is a $(1 - \epsilon) \ln m$-approximation to SET COVER then, NP $\subseteq$ DETERMINISTIC_TIME i.e.,

NP $\subseteq DETERMINISTIC_TIME \left(n^{O(\log \log n)}\right)$, where n is the ground set size.

The set cover problem can be approximated within a ratio of $\log m$[1], where m is the size of the ground set. Arora et al. (1992) and Papadimitriou and Yannakakis(1991) mentions that there is a constant $\delta < 1$ such that it is NP-hard to approximate set cover within a ratio better than δ. Lund and Yannakakis (1994) mentions that it is hard to approximate set cover within a ratio of $\frac{\log_2 m}{2}$.

The aforementioned theorem states that we can not do better than $\log m$ by some multiplicative factor, i.e., for any $\epsilon > 0$, set cover cannot be approximated within a ratio of $(1 - \epsilon) \ln m$ unless NP has a deterministic algorithm with running time $n^{O(\log \log n)}$[1], where n is a ground set size. If such an approximation algorithm exists, then every problem in NP can be solved by a deterministic algorithm running in $n^{O(\log \log n)}$ time.

5 Designing polynomial time approximation schemes(PTAS): Data rounding and Dynamic programming technique

In this section, we will discuss the designing of polynomial time approximation schemes. The two main techniques for polynomial time approximation scheme are *data rounding* and *dynamic programming*. We will consider the KNAPSACK problem to understand polynomial time approximation algorithm. We will design a family of algorithms A_ϵ (PTAS) such that $\{A_\epsilon | \epsilon > 0\}$ and $\epsilon > 0$ of $(1 - \epsilon)$-approximation algorithms for KNAPSACK.

5.1 Running time for PTAS

Let us take a polynomial time algorithm with the running time $O(n^2.n^{1/\epsilon})$. The running time for PTAS, in fact, depends on ϵ. If $\epsilon = 0.5$, the running time will be $O(n^2.n^2)$, which is still polynomial. If ϵ is 0.01, then it will be polynomial ($O(n^2.n^{100})$) with a very high degree, which is not very practical. Therefore, the running time depends polynomially on the input size (n), but arbitrarily on $\frac{1}{\epsilon}$.

5.2 KNAPSACK

"A traveler with a knapsack comes across a treasure hoard. Unfortunately, his knapsack can hold only so much. What items should he place in his knapsack in order to maximize the value of the items he takes away?"[2]

INPUT: A positive integer capacity B (Bound capacity), items i.e., item 1, item 2, ..., item n each having positive integer *size* s_i and positive integer *value* v_i.

OUTPUT: A subset $S \subseteq \{1, 2, 3, ..., n\}$ such that

$\sum_{i \in S} s_i \leq B$

$\max \sum_{i \in S} v_i$

We want to maximize the sum of the values of the item chosen subject to the constraint that the sum of weight of the items chosen is at most the bounding capacity (B).

Comments

1. KNAPSACK is NP-complete, but not strongly NP as it can be solved in psudo-polynomial time.
2. KNAPSACK has an exact algorithm (no approximation required) running in order of n times $\min\{B, V\}$ i.e., $O(n. \min\{B, V\})$
   where $V = \sum_{i=1}^{n} v_i$, n is the number of items.
3. The KNAPSACK problem's running time is not polynomial. In fact, it is *pseudo-polynomial* time. Any algorithm that is polynomial if the number of terms in input is represented in unary is **pseudo-polynomial algorithm**.

5.3 Data Rounding

In the KNAPSACK problem, we need to select a subset of items maximizing the value of the items picked. The item with less value will be removed if the value of the item is only a small fraction, i.e., $0 < \epsilon < 1$ of the optimal value. We are left with an instance where the ratio of the largest and the smallest value (say k) is polynomially bounded i.e for some $\epsilon > 0$, $k = \lfloor \frac{n}{\epsilon} \rfloor$ [3]. Then, by scaling and rounding, we may assume that all numbers are polynomially bounded integers. For

every item i, we then define, new value $\hat{v_i} = \left\lfloor \frac{v_i}{v_{max}}.k \right\rfloor$.

5.4 Dynamic Programming for KNAPSACK

In this section, we will discuss the use of dynamic programming to solve KNAPSACK problem[4].

DEFINITION: Dynamic programming is a method to find the optimal solution of a complex problem by breaking down the problem into smaller sub-problems and solving each sub-problem once using memoization[5].

IDEA: We break down the KNAPSACK problem into sub-problems, as shown in Figure2. The sub-problems are further broken into smaller sub-problems unless we get sub-problem that can be solved easily. Once we know the solutions to all the sub-problems, we can easily obtain the solution to the original problem. Let us consider the two possible cases for the KNAPSACK problem, depending on whether the *item n* is in optimal solution or not. We then solve these two cases in recursion.

Considering the case where *item n* is in the optimal solution, using simple recursion, we solve this sub-problem where the input will contain $item1$, $item2$, ..., $itemn-1$ and the bounding capacity will be $B - s_n$. Similarly, we solve for the case where *item n* is not in the optimal solution, using simple recursion, where input will contain $item1$, $item2$, ..., $itemn-1$ and the bounding capacity will be B. We obtain the maximum value that can be obtained from n items as the maximum of the values obtained from the aforementioned cases.

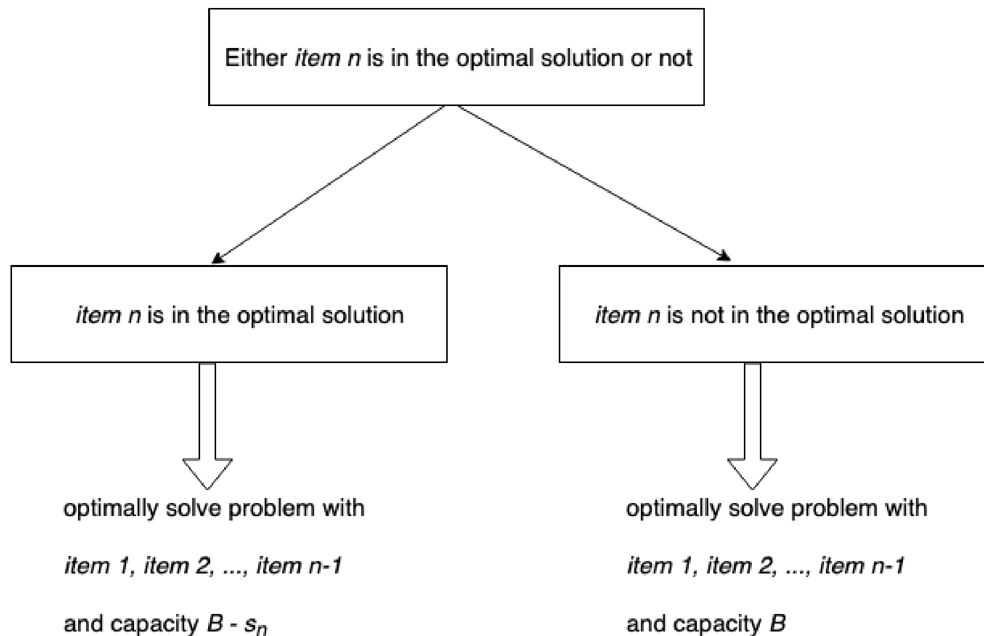

Figure 2: Dynamic Programming for KNAPSACK

Figure 3: Dynamic programming for KNAPSACK: Bottom up computation

The above mentioned idea leads to the recurrence relation shown in Equation 1.

We will have sub-problems for each item. We will solve this using bottom up approach as shown in Figure3. We will start from $j = 0$ and $C = 0$ and find optimal for $j = n$ and $C = B$.

Let, $OPT(j, C)$ denote value of an optimal solution for the sub-problem with items $\{1, , 2, 3, ..., j\}$ and capacity C provided $0 \leq j \leq n$ and $0 \leq C \leq B$.

- If *item* j is in the optimal solution, we solve the sub-problem as $OPT(j - 1, C - s_j) + v_j$
- If *item* j is not in the optimal solution, we solve the sub-problem as $OPT(j - 1, C - s_j)$

For $j > 0, C > 0$,

$$OPT(j, C) = \max\{OPT(j - 1, C - s_j) + v_j, OPT(j - 1, C)\} \quad (1)$$

$OPT(j, C) = 0$ if $j = 0$ or $C = 0$ //base case of our recurrence

References

[1] Feige, Uriel. "A threshold of ln n for approximating set cover." Journal of the ACM (JACM) 45.4 (1998): 634-652.

[2] Williamson, Shmoys, "The Design of Approximation Algorithms"

[3] Gupta, Kedia, "Approximations Algorithms: Dynamic Porgramming" Available at: https://www.cs.cmu.edu/afs/cs/academic/class/15854-f05/www/scribe/lec10.pdf

[4] Kleinberg, Tardos, “Algorithm Design“

[5] https://en.wikipedia.org/wiki/Dynamic_programming